OpenMP for Networks of SMPs

Y. Charlie Hu†, Honghui Lu‡, Alan L. Cox† and Willy Zwaenepoel†

† Department of Computer Science
‡ Department of Electrical and Computer Engineering
Rice University, Houston, Texas 77005
{ychu, hhl, alc, willy}@cs.rice.edu

Abstract

In this paper, we present the first system that implements OpenMP on a network of shared-memory multiprocessors. This system enables the programmer to rely on a single, standard, shared-memory API for parallelization within a multiprocessor and between multiprocessors. It is implemented via a translator that converts OpenMP directives to appropriate calls to a modified version of the TreadMarks software distributed memory system (SDSM). In contrast to previous SDSM systems for SMPs, the modified TreadMarks uses POSIX threads for parallelism within an SMP node. This approach greatly simplifies the changes required to the SDSM in order to exploit the intra-node hardware shared memory.

We present performance results for six applications (SPLASH-2 Barnes-Hut and Water, NAS 3D-FFT, SOR, TSP and MGS) running on an SP2 with four four-processor SMP nodes. A comparison between the threaded implementation and the original implementation of TreadMarks shows that using the hardware shared memory within an SMP node significantly reduces the amount of data and the number of messages transmitted between nodes, and consequently achieves speedups up to 30% better than the original versions. We also compare SDSM against message passing. Overall, the speedups of multithreaded TreadMarks programs are within 7–30% of the MPI versions.

1. Introduction

The OpenMP Application Programming Interface (API) is an emerging standard for parallel programming on shared-memory multiprocessors. It defines a set of program directives and a library for run-time support that augment standard C/C++ [14] and Fortran 77/90 [13]. In contrast to POSIX threads, another shared-memory API, and MPI, a message-passing API, OpenMP facilitates an incremental approach to the parallelization of sequential programs. In other words, the programmer can add a parallelization directive to one loop or subroutine of the program at a time. This is a major reason for OpenMP's growing popularity.

This paper reports on the first system that implements OpenMP on a *network* of shared-memory multiprocessors. This system enables the programmer to rely on a single, standard, shared-memory API for parallelization within a multiprocessor *and* between multiprocessors. Previously, the only standard APIs available on this type of platform were message-passing standards. In our system, OpenMP's program directives are processed by a source-to-source translator that is constructed from the SUIF Toolkit [1]. In effect, the translator converts each OpenMP directive into the appropriate calls to a modified version of the TreadMarks software distributed shared-memory system (SDSM) [2]. The translated source is a standard C or Fortran 77 program that is compiled and linked with the modified TreadMarks system.

In its simplest form, running TreadMarks on a network of SMPs could be achieved by simply executing a (Unix) process on each processor of each multiprocessor node, and have all of these processes communicate through message passing. This approach requires no changes to TreadMarks, and we will therefore refer to it as the *original version*. This version, however, fails to take advantage of the hardware shared memory on the multiprocessor nodes. In order to overcome this limitation, we have built a new version of TreadMarks, in which we use POSIX threads to implement parallelism within a multiprocessor. As a result, the OpenMP threads within a multiprocessor share a single address space. We will refer to this system as the *thread version*. Our approach is distinct from previous SDSM systems for networks of SMPs, like Cashmere-2L [12] and HLRC-SMP [10], which use (Unix) processes to implement parallelism within an SMP. Each of the processes has a separate address space, although the shared memory regions (and some other data structures) are mapped shared between the processes. We will refer to such a system as a *process* *version*.

The use of a single address space within a multiprocessor has plusses and minuses. On the positive side, it reduces the number of changes to TreadMarks to support multithreading on a multiprocessor. For example, the data within an address space on a multiprocessor is shared by default. Furthermore, a page protection operation by one thread applies to the other threads within the same multiprocessor; the operating system maintains the coherence of the page mappings automatically.

On the negative side, using a single address space within a multiprocessor makes it more difficult to provide uniform sharing of memory both between threads on the same node and threads on different nodes. Under POSIX threads, an application's global variables are shared between threads within a multiprocessor, but under TreadMarks they are private with respect to threads on a different multiprocessor.[1] However, since we provide the OpenMP API to the programmer, these differences are hidden by our OpenMP translator.

We measure our OpenMP system's performance on an IBM SP2 with multiprocessor nodes. The machine has four nodes, and each node has four processors. We use six applications: SPLASH-2 Barnes-Hut and Water, NAS 3D-FFT, Red-Black SOR, TSP, and MGS. We compare the results for our OpenMP system to two alternatives: MPI and OpenMP with the original TreadMarks. In both of these cases, the shared memory within a multiprocessor is simply used for fast message passing.

Our results show that using hardware shared memory within an SMP node significantly reduces the amount of data and the number of messages transmitted. Consequently, the speedups improve up to 30% over the original TreadMarks implementation. In addition, we found that the multithreaded TreadMarks performs fewer page protection operations. The reduction in the number of page protection operations ranges from a low factor of 1.9 to a high factor of 6.2. Our experiments also show that the multithreaded TreadMarks programs incur 1.2–5 times fewer page faults than their single-threaded counterparts.

We also compare SDSM against message passing. Overall, the speedups of multithreaded TreadMarks programs are within 7–30% of the MPI versions.

The remainder of this paper is organized as follows. Section 2 presents an overview of the OpenMP API. Section 3 presents an overview of the original TreadMarks system and describes the modifications to support OpenMP and a network of shared-memory multiprocessors. Section 4 describes the source-to-source translator for OpenMP. Section 5 evaluates our system's overall performance and compares it to MPI and TreadMarks without support for shared-memory multiprocessors. Section 6 discusses related work. Section 7 summarizes our conclusions.

2. The OpenMP API

The OpenMP API [13, 14] defines a set of program directives that enable the user to annotate a sequential program to indicate how it should be executed in parallel. In C/C++, the directives are implemented as `#pragma` statements, and in Fortran 77/90 they are implemented as comments. OpenMP is based on a fork-join model of parallel execution. The sequential code sections are executed by a single thread, called the *master thread*. The parallel code sections are executed by all threads, including the master thread. OpenMP provides three kinds of directives: parallelism/work sharing, data environment, and synchronization. We only explain the directives relevant to this paper, and refer interested readers to the OpenMP standard [13, 14] for the full specification.

The fundamental directive for expressing parallelism is the `parallel` directive. It defines a *parallel region* of the program, that is executed by multiple threads. All of the threads perform the same computation, unless a *work sharing* directive is specified within the parallel region. Work sharing directives, such as `for`, divide the computation among the threads. For example, the `for` directive specifies that the iterations of the associated loop should be divided among the threads so that each iteration is performed by a single thread. The `for` directive can take a `schedule` clause that specifies the details of the assignment of the iterations to threads. Schedules can specify assignments such as round-robin or block. OpenMP also defines shorthand forms for specifying a parallel region containing a single work sharing directive. For example, the `parallel for` directive is shorthand for a `parallel` region that contains a single `for` directive.

The data environment directives control the sharing of program variables that are defined outside of a parallel region.[2] They appear at the beginning of a parallel region, immediately following the parallel directives. The data environment directives include: `shared`, `private`, `firstprivate`, `reduction`, and `threadprivate`. Each directive is followed by a list of variables. Variables default to `shared`, which means shared among all the threads in a parallel region. A `private` variable has a separate copy per thread. Its value is undefined when entering or exiting a parallel region. A `firstprivate` variable has the same attributes as a `private` variable except that the private copies are initialized to the variable's value at the time the parallel region is entered. The `reduction` directive identifies reduction variables. According to the

[1] This issue is not specific to TreadMarks. The use of a single address space on a node would give rise to similar non-uniformities in sharing within and across nodes with other SDSM systems.

[2] The variables defined inside of a parallel region are implicitly private.

standard, reduction variables must be scalar, but we extend the standard to include arrays. Finally, OpenMP provides the `threadprivate` directive for named common blocks in Fortran 77/90 and global variables in C/C++. `Threadprivate` variables are private to each thread, but they are global in the sense that they are defined for all parallel regions in the program, unlike `private` variables which are defined only for a particular parallel region.

The synchronization directives include `barrier`, `critical`, and `flush`. A `barrier` directive causes the thread to wait until all of the other threads in the parallel region have reached this point. After the `barrier`, all threads are guaranteed to see all modifications made before the barrier. A `critical` directive restricts access to the enclosed code to only one thread at a time. When a thread enters a critical section, it is guaranteed to see all modifications made by all the threads that entered the critical section earlier. The `flush` directive specifies a "cross thread" sequence point in the program at which all threads are guaranteed to have a consistent view of the variables named in the `flush` directive, or of all of the memory if no variables are specified.

3. TreadMarks

TreadMarks [2] is a user-level SDSM system that runs on most Unix and Windows NT-based systems. It provides a global shared address space on top of physically distributed memories. The parallel threads synchronize via primitives similar to those used in hardware shared-memory machines: barriers and locks. In C, the program has to call the `Tmk_malloc` routine to allocate shared variables in the shared heap. In Fortran, the shared data are placed in a common block loaded in a standard location.

3.1. Implementation Overview

Memory coherence and synchronization are the key functions performed by TreadMarks.

3.1.1 Memory Coherence

TreadMarks relies on user-level memory management support provided by the operating system to detect accesses to shared memory at the granularity of a page. A *lazy invalidate* version of *release consistency* (RC) [7] and a multiple-writer protocol are employed to reduce the amount of communication involved in implementing the shared memory abstraction.

RC is a relaxed memory consistency model. In RC, *ordinary* shared memory accesses are distinguished from *synchronization* accesses, with the latter category divided into *acquire* and *release* accesses. RC requires ordinary shared memory updates by a thread p to become visible to another thread q only when a subsequent release by p becomes visible to q via some chain of synchronization events. In practice, this model allows a thread to buffer multiple writes to shared data in its local memory until a synchronization point is reached.

With the multiple-writer protocol, two or more threads can simultaneously modify their own copies of a shared page. Their modifications are merged at the next synchronization operation in accordance with the definition of RC, thereby reducing the effect of false sharing.

The *lazy* implementation delays the propagation of consistency information until the time of an acquire. Furthermore, the releaser informs the acquiring thread which pages have been modified, causing the acquiring thread to *invalidate* its local copies of these pages. A thread incurs a page fault on the first access to an invalidated page, and obtains an up-to-date version of that page from the previous releasers.

3.1.2 Synchronization

Barrier arrivals are modeled as releases, and barrier departures are modeled as acquires. Barriers have a centralized manager. At a barrier arrival, each thread sends a release message to the manager, and waits for a departure message. The manager broadcasts a barrier departure message to all threads after all have arrived at the same barrier.

The two primitives for mutex locks are lock release and lock acquire. Each lock has a statically assigned manager. The manager records which thread has most recently requested the lock. All lock acquire requests are sent to the manager, and, if necessary, forwarded by the manager to the thread that last requested the lock.

3.2. Modifications for OpenMP

To support OpenMP-style environments, recent versions of TreadMarks include `Tmk_fork` and `Tmk_join` primitives, specifically tailored to the fork-join style of parallelism expected by OpenMP and most other shared memory compilers [1]. For performance reasons, all threads are created at the start of a program's execution. During sequential execution, the slave threads are blocked waiting for the next `Tmk_fork` issued by the master.

3.3. Modifications for Networks of Multiprocessors

The modified version of TreadMarks uses POSIX threads to implement parallelism within a multiprocessor. Hence, the OpenMP threads within a multiprocessor share a *single* address space. This has many advantages and a few disadvantages in both the implementation of TreadMarks and its interface.

3.3.1 Implementation Issues

By using POSIX threads, data is shared by default among the processors within a single machine, and coherence is maintained automatically by the hardware. Thus, we did not have to modify TreadMarks to enable the sharing of application data or its own internal data structures between processors within the same machine. We did, however, have to modify TreadMarks to place some data structures, such as message buffers, in thread-private memory.

Synchronizing access by the processors within a machine to the internal data structures was straightforward. The critical sections within TreadMarks were already guarded by synchronization because incoming data and synchronization requests occur asynchronously, interrupting the application. Thus, with one exception, we simply changed the existing synchronization to work with POSIX threads. The exception is that we added a per-page mutex to allow greater concurrency in the page fault handler.

The synchronization functions provided by TreadMarks to the program were modified to combine the use of POSIX threads-based synchronization between processors within a machine and the existing TreadMarks implementation between machines. Thus, the program can continue to use a single API, that of TreadMarks, for synchronization.

Our last change to the implementation was in the memory coherence mechanism. We added a second mapping at a different address within each machine's address space for the TreadMarks supported shared data heap/common block, i.e., the memory that is shared between machines. The first, or original, mapping is used exclusively by the application; the modified version of TreadMarks never accesses the shared data through this mapping. Instead, it uses the second mapping, which permits read and write access at all times. This mapping is used to update shared-memory pages so that the application's mapping can remain invalid while the update is in progress. This insures that another thread cannot read or modify the page until the update is complete.

In fact, the use of two mappings reduces the number of `mprotect`, or page protection, operations performed by TreadMarks, even on a single-processor node. For example, in the original TreadMarks, a read access to an invalid page would result in two `mprotect` operations: one to enable write access in order to update the page and another to make the page read-only after the update. In the modified version, only the latter `mprotect` operation is performed. The second mapping eliminates the need for the first `mprotect` operation.

Finally, because of our use of a single address space, the operating system automatically maintains the coherence of the page mappings in use by the different processors within a machine. Furthermore, an `mprotect` by one thread within a machine applies to the other threads. In contrast, systems such as Cashmere-2L [12], that use Unix processes instead of POSIX threads, must perform the same `mprotect` in each process's address space. The reason is that `mprotect` only applies to the calling process's address space, even if the underlying memory is shared between address spaces.

3.3.2 Interface Issues

Our use of POSIX threads had one undesirable effect on the TreadMarks interface. Under POSIX threads, global variables are shared; whereas, in the original TreadMarks API, global variables are private. Thus, in our modified version of TreadMarks, global variables are shared between threads within a multiprocessor but are private with respect to threads on a different multiprocessor. Rather than attempting to solve this problem in the run-time system, we chose to address it in the OpenMP translator where a solution is straightforward.

4. The OpenMP Translator

The OpenMP to TreadMarks translation process is relatively simple, because TreadMarks already provides a shared memory API on top of a network of computers. First, the OpenMP synchronization directives translate directly to TreadMarks synchronization operations. Second, the compiler translates the code sections marked with `parallel` directives to fork-join code. Third, it implements the data environment directives in ways that work with both TreadMarks and POSIX threads, hiding the interface issues discussed in Section 3.3.2 from the programmer.

4.1. Implementing Parallel Directives

To translate a sequential program annotated with parallel directives into a fork-join parallel program, the translator encapsulates each parallel region into a separate subroutine. This subroutine also includes code, generated by the compiler, that allows each thread to determine, based on its thread identifier, which portions of a parallel region it needs to execute. At the beginning of a parallel region, the master thread passes a pointer to this subroutine to the slave threads at the time of the fork. Pointers to `shared` variables and initial values of `firstprivate` variables are copied into a structure and passed to the slaves at the fork.

4.2. Implementing Data Environment Directives

Variables accessed within a parallel region default to `shared`. If a global variable is annotated `threadprivate`, it cannot be annotated again within a parallel region. Thus, the translator allocates all global variables on the shared heap unless they are annotated `threadprivate`.

For each `threadprivate` global variable, the compiler allocates an array of nt copies of the global variable, where nt is the number of threads per node. Each reference to the global variable is replaced by a reference to the array, specifically, a reference to the element corresponding to the thread's (local) id.

In TreadMarks, a thread's stack is kept in private memory. Thus, variables declared within a procedure that are accessed within a parallel region must be moved to the shared heap. In addition, variables declared within a procedure and passed by reference to another procedure are moved to the shared heap because the translator cannot prove that such a variable will not be used in a parallel region. Storage for these variables is allocated at the beginning of the procedure and freed at the end.

Implementing `private` variables is straightforward: whenever a variable is annotated private within a parallel region, it is redeclared in the procedure generated by the compiler that encapsulates the parallel region. Because each thread calls this procedure after the fork, these variables will be allocated on the private stack of each thread.

5. Performance

5.1. Platform

Our experimental platform is an IBM SP2 consisting of four SMP nodes. Each node contains four IBM PowerPC 604 processors and 1 Gbyte of memory. All of the nodes are running AIX 4.2.

5.2. Applications and Their OpenMP Implementations

We use six applications in this study: SPLASH-2 Barnes-Hut, NAS 3D-FFT, SPLASH-2 Water, Red-Black SOR, TSP and MGS. Table 1 summarizes the problem sizes, the sequential running times, and the parallelization directives used in the OpenMP implementations of the applications. The sequential running times are used as the basis for the speedups reported in the next section.

Appl.	Size, Iterations	Sequential Time (sec.)	OpenMP Parallel Directives
Barnes	65536	158.0	parallel region
3D-FFT	$128\times128\times64$, 10	65.2	parallel for
Water	4096, 4	760.3	parallel for/region
SOR	8K x 4K, 20	149.0	parallel for
TSP	19 cities, -r14	248.1	parallel region
MGS	2K x 2K	563.3	parallel for

Table 1. Application, problem size, sequential execution time, and parallelization directive(s) in the OpenMP programs.

Barnes Barnes-Hut from SPLASH-2 [15] is an N-body simulation code using the hierarchical Barnes-Hut method. A shared tree structure is used to represent the recursively decomposed subdomains (cells) of the three-dimensional physical domain containing all of the particles. The other shared data structure is an array of particles corresponding to the leaves of the tree. Each iteration is divided into two steps.

1. Tree building: A single thread reads the particles and rebuilds the tree.
2. Force evaluation: All threads participate. First, they divide the particles by traversing the tree in the Morton ordering (a linear ordering of the points in higher dimensions) of the cells. Specifically, the ith thread locates the ith segment. The size of a segment is weighted according to the workload recorded from the previous iteration. Then, each of the threads performs the force evaluation for its particles. This involves a partial traversal of the tree. Overall, each thread reads a large portion of the tree.

In OpenMP, the force evaluation is parallelized using the `parallel` region directive.

3D-FFT 3D-FFT from the NAS benchmark suite [3] solves a partial differential equation using three-dimensional forward and inverse FFT. The program has three shared arrays of data elements and an array of checksums. The computation is decomposed so that every iteration includes local computation and a global transpose, with both expressed as data parallel operations.

In OpenMP, the data parallelism is expressed using the `parallel for` directive.

Water Water from the SPLASH-2 [15] benchmark suite is a molecular dynamics simulation. The main data structure in Water is a one-dimensional array of molecules. During each time step, both intra- and inter-molecular potentials are computed. The parallel algorithm statically divides the array of molecules into equal sized contiguous blocks, assigning each block to a thread. The bulk of the interprocessor communication results from synchronization that takes place during the intermolecular force computation.

In OpenMP, the evaluation of intra-molecule potentials requires no interactions between molecules and is parallelized using the `parallel for` directive. The evaluation of inter-molecule potentials is parallelized using the `parallel` region directive. Each thread is assigned a subset of the molecules. It accumulates the results of the force computation into private memory during the computation, and only synchronizes with the other threads afterwards to perform a reduction.

SOR Red-Black Successive Over-Relaxation is a method for solving partial differential equations by iterating over a two-dimensional array. In every iteration, each of the array elements is updated to the average of the element's four nearest neighbors.

These data parallel operations are expressed in OpenMP using the `parallel for` directive.

TSP TSP solves the traveling salesman problem using a branch-and-bound algorithm. The major data structures are a pool of partially evaluated tours, a priority queue containing pointers to tours in the pool, a stack of pointers to unused tour elements in the pool, and the current shortest path. A thread repeatedly dequeues the most promising path from the priority queue, either extends it by one city and enqueues the new path, or takes the dequeued path and tries all permutations of the remaining cities.

In OpenMP, the threads are created using the `parallel` region directive. Accesses to the priority queue are synchronized using the `critical` directive.

MGS Modified Gramm-Schmidt (MGS) computes an orthonormal basis for a set of N-dimensional vectors. At the ith iteration, the algorithm first normalizes the ith vector sequentially, then makes all vectors $j > i$ orthogonal to vector i in parallel. Vectors are assigned to threads in a cyclic manner to balance the load. All threads synchronize at the end of each iteration.

In OpenMP, the normalization of each vector is performed by the master thread, and the parallel updates are expressed using the `parallel for` directive with a static schedule. The static schedule uses a chunk size of one.

5.3. Results

We first compare the performance of the OpenMP programs translated into TreadMarks programs modified to use POSIX threads within an SMP node (OpenMP/thread) against the performance of those same programs translated into original TreadMarks programs using processes (OpenMP/original). In the latter, processes on the same node communicate via message passing instead of using the hardware shared memory.

We then compare the OpenMP/thread and OpenMP/original versions of the applications against MPI versions of the same applications. We use the MPICH (http://www.mcs.anl.gov/mpi/mpich) implementation of MPI, because it takes advantage of the hardware shared memory when sending messages within the same node. We count both the total number of messages and the number of messages that actually cross node boundaries.

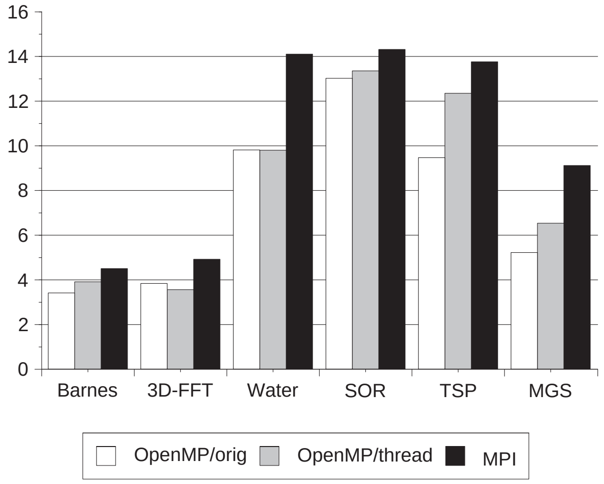

Figure 1. Speedup comparison between the OpenMP/original, OpenMP/thread, and MPI versions of the applications on an SP2 with four four-processor SMP nodes.

Figure 1 shows the speedups for the OpenMP/original programs with four processes per node, OpenMP/thread programs with four threads per node, and MPI programs with four processes per node on the four-node SP2. Table 2 compares the amount of data and the number of messages communicated for the different cases.

5.3.1 OpenMP/original versus OpenMP/thread

In terms of relative speedup, the applications can be categorized into three groups. The first group consists of TSP and MGS, which have a low to moderate computation to communication ratio. Thus, the five-fold reduction in the amount of data transmitted results in significant speedups. The second group consists of Barnes, Water, and SOR, which have very high computation to communication ratios. In this case, a 3.3 to 9 fold reduction in data leads to little improvement in running time. FFT forms the third group, where we see a slight slowdown for the OpenMP/thread code. The slowdown happens in the transpose stage where all processors request data from one processor at a time, invoking large numbers of request handlers on that processor's node. We suspect that AIX 4.2 incurs a certain degree of serialization in handling these requests which contributed

Appl.	OpenMP/ original	OpenMP/ thread	MPI Total	MPI Off-node
Data (Mbytes)				
Barnes	543.0	166.4	259.7	207.8
3D-FFT	159.4	126.5	157.3	125.8
Water	192.3	42.7	34.6	26.0
SOR	0.64	0.07	9.8	2.0
TSP	2.8	0.55	0.03	0.026
MGS	508.6	102.2	251.6	201.3
Messages				
Barnes	841565	100259	720	576
3D-FFT	40975	31694	9750	7800
Water	78402	24667	1776	1344
SOR	3637	735	1200	240
TSP	9227	4853	1256	1070
MGS	184583	37041	30720	24576

Table 2. Amount of data and number of messages transmitted in the OpenMP/original, OpenMP/thread, and MPI versions of the applications on an SP2 with four four-processor SMP nodes.

Application	Orig/1	Thrd/1	Orig/4	Thrd/4
`mprotect` Count				
Barnes	67315	43158	201797	84843
3D-FFT	65650	50200	97690	50588
Water	32668	23073	119970	34049
SOR	1209	969	6037	969
TSP	6947	5529	11628	5438
MGS	30730	21511	86154	21118
Page Fault Count				
Barnes	25882	25882	97148	68205
3D-FFT	30860	30860	39020	31155
Water	13533	13523	46130	30329
SOR	480	480	2400	480
TSP	2895	2889	4794	4047
MGS	14336	14336	40346	32404
diff Count				
Barnes	14984	14984	44028	33253
3D-FFT	15404	15404	19370	15501
Water	5090	5090	13017	7890
SOR	240	240	1200	240
TSP	1394	1394	1599	1357
MGS	3072	3072	3827	3724

Table 3. Number of `mprotect` operations, page faults, and diffs in the OpenMP/original and OpenMP/thread versions of the applications. Orig/1 and Orig/4 denote OpenMP/original with 1 and 4 processes on a node, and Thrd/1 and Thrd/4 denote OpenMP/thread with 1 and 4 threads on a node, respectively.

to the slowdown of the thread version. The reduction in data and number of messages of the thread version of 3D-FFT is too small to offset the slowdown from the above serialization.

Overall, compared to OpenMP/original, the OpenMP/thread programs send less data, from a low of 26% less data for 3D-FFT to a high of 9.1 times less data for SOR, and fewer messages, from a low of 29% fewer messages for 3D-FFT to a high of 8.4 times fewer messages for Barnes.

Table 3 compares the number of times that the `mprotect` operation is performed in the process and the thread versions of the translated OpenMP programs on four SMP nodes. First, the OpenMP/thread programs with one thread per node perform 25–56% fewer `mprotect` operations than the corresponding OpenMP/original versions with one process per node, indicating that the alias mapping (See Section 3.3.1) reduces the number of `mprotect` operations independent of any multithreading effects. Second, the OpenMP/thread programs with four threads per node perform 1.9–6.2 times fewer `mprotect` operations than the OpenMP/original codes with four processes per node, indicating that multithreading further reduces the number of `mprotect` operations.

Table 3 also shows that multithreading can reduce the number of page faults: while the number of page faults incurred by OpenMP/thread with one thread per node and OpenMP/original with one process per node are the same, the OpenMP/thread programs with four threads per node incur 1.2–5 times fewer page faults than their OpenMP/original counterparts with four processes per node. This reduction comes from two sources. First, for multiple-reader pages, only one of the threads on a node needs to fault in order to update the page and make it accessible by all of the threads, whereas each of the processes on a node has to fault once to update its own copy. Second, using multithreading eliminates the faults required by a process accessing a page that was invalidated by other processes on the same node.

Finally, Table 3 shows that for 16-way parallelism on four nodes, OpenMP/thread creates 1.03–5 times fewer diffs than OpenMP/original.

5.3.2 OpenMP versus MPI

A previous study comparing SDSM with message passing [8] has shown that, in general, SDSM programs send more messages and data than message passing versions due to the separation of synchronization and data transfer, the need to handle access misses caused by the use of an invalidate protocol, false sharing, and diff accumulation for migratory data. In our experiments, the OpenMP/original programs sent between 3 and 1169 times more messages than their MPI counterparts. The difference was the least for SOR and the most for Barnes. The reason that the MPI version of Barnes sends so few messages is because it replicates the particles and duplicates rebuilding the tree by every process. As a consequence, within an iteration, the only communication by each process is a single broadcast of all the particles modified by that process. Except for SOR, the amount of data sent by TreadMarks ranges from an equal amount for 3D-FFT to 93 times more data for TSP. For SOR, because a large percentage of the elements remain unchanged, and because TreadMarks only communicates diffs, the TreadMarks program sends 15.5 times less data than the MPI code, which always communicates whole boundary rows. As has been demonstrated by Dwarkadas et al. [5], many of the causes of the gap in data and message count between SDSM and MPI can be overcome with additional compiler support, which is currently not present in our translator.

Our results with the OpenMP/thread programs show that on SMP nodes using multithreading in SDSM can significantly reduce the above gaps in the number of messages and the amount data transmitted between SDSM and MPI programs. In fact, the OpenMP/thread programs send from 1.63 times fewer messages, for SOR, to only 139 times more messages, for Barnes, than the MPI codes. Similarly, OpenMP/thread sends 1.2–140 times less data than MPI for four out of the six applications, and only 1.2–18 times more data than MPI for the other two.

Table 2 further shows that, for all applications except SOR, the MPI versions send about $\frac{12}{15}$ of the total data and messages across node boundaries. This corresponds to the ratio of off-node processors versus all processors as viewed by each processor. For SOR, the MPI program sends only 20% of the total data and messages across node boundaries because communication only occurs between neighboring processes and the neighboring processes are in most cases within the same node.

6. Related Work

Previously, we developed support for OpenMP programming on networks of single-processor workstations through a compiler that targets the TreadMarks software distributed shared-memory system [9]. Our experiments showed that the OpenMP versions of the five selected applications achieve performance within 17% of their handwritten TreadMarks counterparts, suggesting that the compiler and the fork-join model incur very little overhead.

We are aware of four implementations of SDSM on networks of SMPs [6, 11, 12, 10].

The SMP-Shasta system [11] implements eager release consistency (ERC) and a single-writer protocol with variable granularity by instrumenting the executable to insert access control operations (in-line checks) before shared memory references. Novel techniques are developed to minimize the overhead of in-line checks.

In their paper, Erlichson et al. [6] present a single-writer sequential consistency implementation, and identify network bandwidth as the bottleneck. Earlier work (e.g. [4]) has demonstrated that the performance of such a system can be poor when false sharing occurs. Finally, our implementation is a relatively portable user-level implementation, while theirs is a kernel implementation specific to the Power Challenge Irix kernel.

Cashmere-2L [12] uses Unix processes instead of POSIX threads. Therefore, it must perform the same `mprotect` in each process's address space. The reason is that `mprotect` only applies to the calling process's address space, even if the underlying memory is shared between address spaces. The protocol implemented by Cashmere-2L also takes advantage of the Memory Channel network interface unique to the DEC Alpha machines.

Samanta et al. [10] present an implementation of a lazy, home-based, multiple-writer protocol across SMP nodes. Similar to Cashmere-2L, their implementation uses Unix processes instead of POSIX threads in exploiting the hardware coherence and synchronization within an SMP.

7. Conclusions

In this paper, we present the first system that implements OpenMP on a *network* of shared-memory multiprocessors. This system enables the programmer to rely on a single, standard, shared-memory API for parallelization within a multiprocessor *and* between multiprocessors. The system is implemented via a translator that converts OpenMP directives to appropriate calls to a modified version of TreadMarks that exploits the hardware shared memory within an SMP node using POSIX threads.

Using the hardware shared memory within an SMP node can significantly reduce data and messages transmitted by a SDSM. In our experiments, the translated multithreaded TreadMarks codes send from a low of 26% less data and 29% fewer messages to a high of 9.1 times less data and 8.4 times fewer messages for our collection of applications than the translated single-threaded TreadMarks counterparts. As a consequence, they achieve up to 30% better speedups than the latter for all applications except 3D-FFT, for which the thread version is 8% slower than the process version. We suspect the slowdown in 3D-FFT is due to an artifact of AIX and not indicative of a more general problem. Overall, the speedups of multithreaded TreadMarks codes on four four-way SMP SP2 nodes are within 7–30% of the MPI versions.

References

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, Feb. 1995.

[2] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[3] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report TR RNR-91-002, NASA Ames, Aug. 1991.

[4] J. Carter, J. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related information in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, Aug. 1995.

[5] S. Dwarkadas, A. Cox, and W. Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 186–197, Oct. 1996.

[6] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the performance of clustered distributed virtual shared memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[7] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[8] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Quantifying the performance differences between PVM and TreadMarks. *Journal of Parallel and Distributed Computing*, 43(2):56–78, June 1997.

[9] H. Lu, Y. C. Hu, and W. Zwaenepoel. OpenMP on networks of workstations. In *Proceedings of Supercomputing '98*, Nov. 1998.

[10] R. Samanta, A. Bilas, L. Iftode, and J. Singh. Home-based SVM protocols for SMP clusters: design and performance. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Feb. 1998.

[11] D. Scales, K. Gharachorloo, and A. Aggarwal. Fine-grain software distributed shared mmeory on SMP clusters. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 125–136, Feb. 1998.

[12] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software coherent shared memory on a clustered remote write network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 170–183, Oct. 1997.

[13] The OpenMP Forum. OpenMP Fortran Application Program Interface, Version 1.0. http://www.openmp.org, Oct. 1997.

[14] The OpenMP Forum. OpenMP C and C++ Application Program Interface, Version 1.0. http://www.openmp.org, Oct. 1998.

[15] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.